deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS Lab activities

v2023-03-02

Introductory notes and setup

Work submission

You should create a personal (git) repository for your TQS **individual portfolio** in which you will be including your solutions for the labs (e.g.: **tqs_123567** , the number being your student number). Keep a **clean organization** that maps the exercise structure, e.g.: 🗁lab1/lab1_1; 🗁lab1/lab1_2; 🗁lab2/lab2_1; 🗁lab2/lab2_2…
You are expected to keep your repo (portfolio) up to date and complete. Teachers will select a few exercises later for assessment [not all, but representative samples].

Lab activities

Be sure that your developer environment meets the following requirements:

- Java development environment (JDK; v11 or v17 suggested). Note that you should install it into a path without spaces or special characters (e.g.: avoid \Users\José Conceição\Java).
- Maven configured to run in the command line. Check with:

  ```
  $ mvn --version
  ```

- Java capable IDE, such as IntelliJ IDEA (version “Ultimate” suggested) or VS Code.

Lab 1 Unit testing (with JUnit 5)

Learning objectives

- Identify relevant unit tests to verify the contract of a module.
- Write and execute unit tests using the JUnit framework.
- Link the unit tests results with further analysis tools (e.g.: code coverage)

Key points

- Unit testing is when you (as a programmer) write test code to verify units of (production) code. A unit is a small, coherent subset of a much larger solution. A true "unit" should not depend on the behavior of other (collaborating) modules.
- Unit tests help the developers to (i) understand the module contract (what to construct); (ii) document the intended use of a component; (iii) prevent regression errors; (iv) increase confidence in the code.
- JUnit and TestNG are popular frameworks for unit testing in Java.

> **JUnit best practices: unit test one object at a time**
>
> A vital aspect of unit tests is that they're finely grained. A unit test independently examines each object you create, so that you can isolate problems as soon as they occur. If you put more than one object under test, you can't predict how the objects will interact when changes occur to one or the other. When an object interacts with other complex objects, you can surround the object under test with predictable test objects. Another form of software test, integration testing, examines how working objects interact with each other. See chapter 4 for more about other types

1.1 Stack contract

In this exercise, you will implement a stack data structure (TqsStack) with appropriate unit tests. Be sure to adopt a **write-the-tests-first** workflow:

a) Create a new project (**maven project** for a Java standard application). You may need to update the Java version in the POM.xml and other dependencies. You may "clone" from a sample project:
   - Adapt from the quick start project for Maven[1].

b) Add the required dependencies to run JUnit 5 tests[2]. Example:
   - sample content for POM.xml (note the elements: **junit-jupiter** and **maven-surefire-plugin**)

```
<!-- ... -->
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.9.2</version>
        <scope>test</scope>
    </dependency>
    <!-- ... -->
</dependencies>
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M7</version>
        </plugin>
        <plugin>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>3.0.0-M7</version>
        </plugin>
    </plugins>
</build>
<!-- ... -->
```

c) Create the required class definition (**just the "skeleton"**, do not implement the methods body yet!). The **code should compile**, but the **implementation is yet incomplete** (you may need to add dummy return values).

d) Write unit tests that will verify the TqsStack contract.

   You may use the IDE features to generate the testing class; note that the IDE support will vary. Be sure to use JUnit 5.x. [Mixing JUnit 4 and JUnit 5 dependencies will prevent the test methods to run as expected!]

   Your tests will verify several assertions that should evaluate to true for the test to pass.

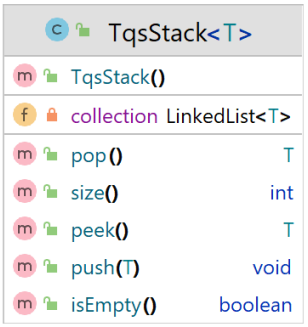
e) Run the tests and prove that TqsStack implementation is not valid yet (the tests should **run** and **fail** for now, the first step in Red-Green-Refactor).

f) Correct/add the missing implementation to the TqsStack;

[1] Delete the "pom-SNAPSHOT.xml", if you are cloning the project to use as a quick starter.
[2] If using IntelliJ: you may skip this step and ask, later, the IDE to fix JUnit imports.

g) Run the unit tests.

h) Iterate from steps d) to f) and confirm that all tests are passing.



Suggested stack contract:

- push(x): add an item on the top
- pop: remove the item at the top
- peek: return the item at the top (without removing it)
- size: return the number of items in the stack
- isEmpty: return whether the stack has no items

What to test[3]:

a) A stack is empty on construction.

b) A stack has size 0 on construction.

c) After n pushes to an empty stack, n > 0, the stack is not empty and its size is n

d) If one pushes x then pops, the value popped is x.

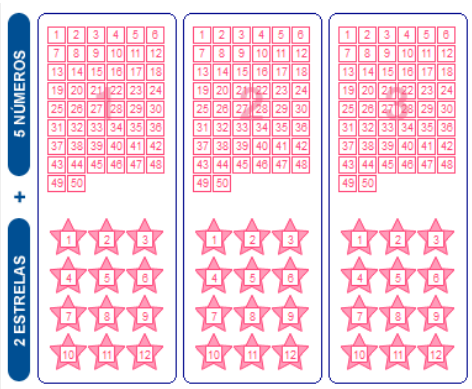
e) If one pushes x then peeks, the value returned is x, but the size stays the same

f) If the size is n, then after n pops, the stack is empty and has a size 0

g) Popping from an empty stack does throw a NoSuchElementException [You should test for the Exception occurrence]

h) Peeking into an empty stack does throw a NoSuchElementException

i) For bounded stacks only: pushing onto a full stack does throw an IllegalStateException



1.2 EuroMillions

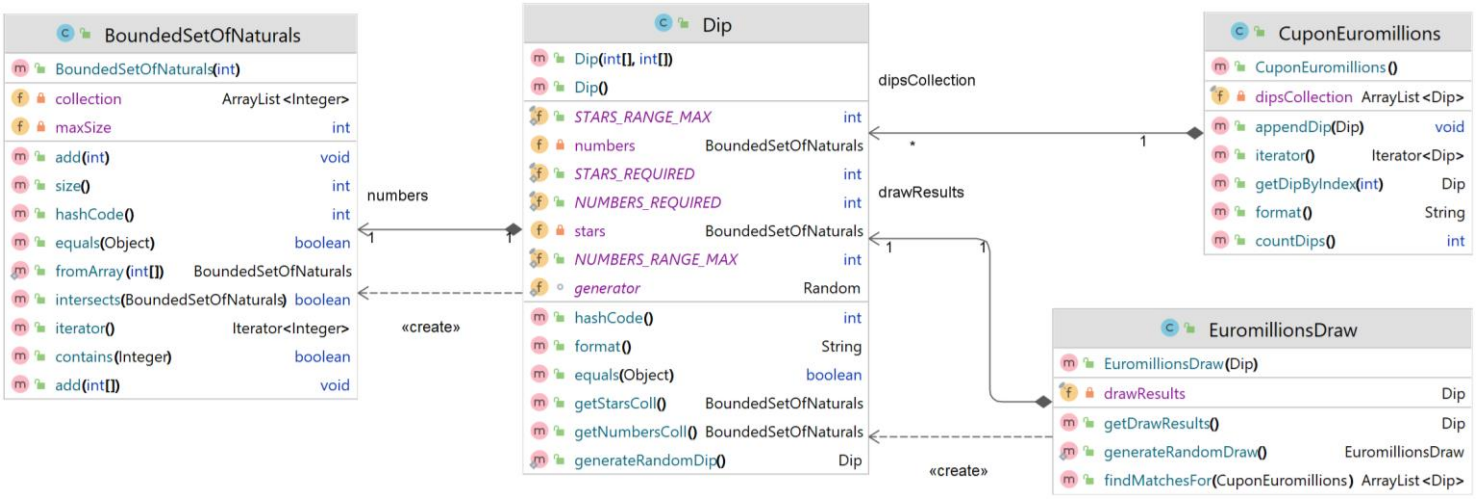
Let us consider the "Euromilhões" use case.

2a/ Pull the "euromillions-play" project
The supporting implementations is visualized in the class diagram that follows.
Get familiar with the solution and existing tests.

Class	Purpose
BoundedSetOfNaturals	Reusable set data structure no duplicates allowed (it is a Set) only natural numbers in the range [1, +∞]. the max size of the set (count of elements) is bounded to a limit allows set operations (contains element?, append element, calculate intersection with another set,…)
Dip	A collection of 5 "numbers" and 2 "stars" (a "column" in the Euromillions playing coupon)
CouponEuromillion	One or more Dips, representing a bet from a player.

[3] Adapted from http://cs.lmu.edu/~ray/notes/stacks/

EuromillionsDraw	Holds the winning dip and can find matched for a given player coupon.



2b/ Make the necessary changed for the existing (non-disabled) unit tests pass.

For the (failing) test:	**You should:**
testConstructorFromBadRanges	Change Dip implementation. Be sure to raise the expected exception if the arrays have invalid numbers (out of range numbers)

Note: you may suspend temporary a test with the @Disabled tag (useful while debugging the tests themselves).

2c/ **Assess the coverage** level in project "Euromillions-play".
Configure the maven project to run Jacoco analysis, if needed.
Run the maven "test" goal and then "jacoco:report" goal. You should get an HTML report under *target/jacoco*.

```
$ mvn clean test jacoco:report
```

Analyze the results accordingly. Which classes/methods offer less coverage? Are all possible [decision] branches being covered?
Collect evidence of the coverage for "BoundedSetOfNaturals".

Note: IntelliJ has an integrated option to run the tests with the coverage checks (without setting the Jacoco plugin in POM). But if you do it at Maven level, you can use this feature in multiple tools.

2c/
Consider the class BoundedSetOfNaturals and its expected contract.
What kind of unit test are worth writing for proper validation of BoundedSetOfNaturals?
Complete the project, adding the tests you have identified. (You may also enhance the implementation of BoundedSetOfNaturals, if necessary.)

2d/
Run Jacoco coverage analysis and compare with previous results. In particular, compare the "before" and "after"for the BoundedSetOfNaturals class.

Troubleshooting some frequent errors

➔ "Test are run from the IDE but not from command line."

Be sure to configure the Surefire plug-in in Maven (example).

Explore

- JetBrains Blog on Writing JUnit 5 tests (with vídeo).
- Book: JUnit in Action. Note that you can access it from the OReilly on-line library.
- Book: "Mastering Software Testing with JUnit 5" and associated GitHub repository with examples
- JUnit 5 cheat sheet.

Lab 2 Mocking dependencies (for unit testing)

Context and key points

Learning objectives

- Prepare a project to run unit tests (JUnit 5) and mocks (Mockito), with mocks injection (@Mock).
- Write and execute unit tests with mocked dependencies.
- Experiment with mock behaviors: strict/lenient verifications, advanced verifications, etc.

Preparation

Get familiar with sections 1 to 3 in the Mockito (Javadoc) documentation.

Explore

- There is a recent book on JUnit and Mockito available from OReilly. The lessons are available as short videos too.

2.1 Stocks portfolio

Consider the example in Figure 1: the StocksPortfolio holds a collection of Stocks; the current value of the *portfolio* depends on the current condition of the *Stock Market*. **StockPortfolio#getTotalValue()** method calculates the value of the portfolio (by summing the current value of owned stock, looked up in the stock market service).

1a/

Implement (at least) one test to verify the implementation of **StockPortfolio#getTotalValue().**
Given that test should have predictable results, you need to address the problem of having non-deterministic answers from the **IStockmarketService** interface.

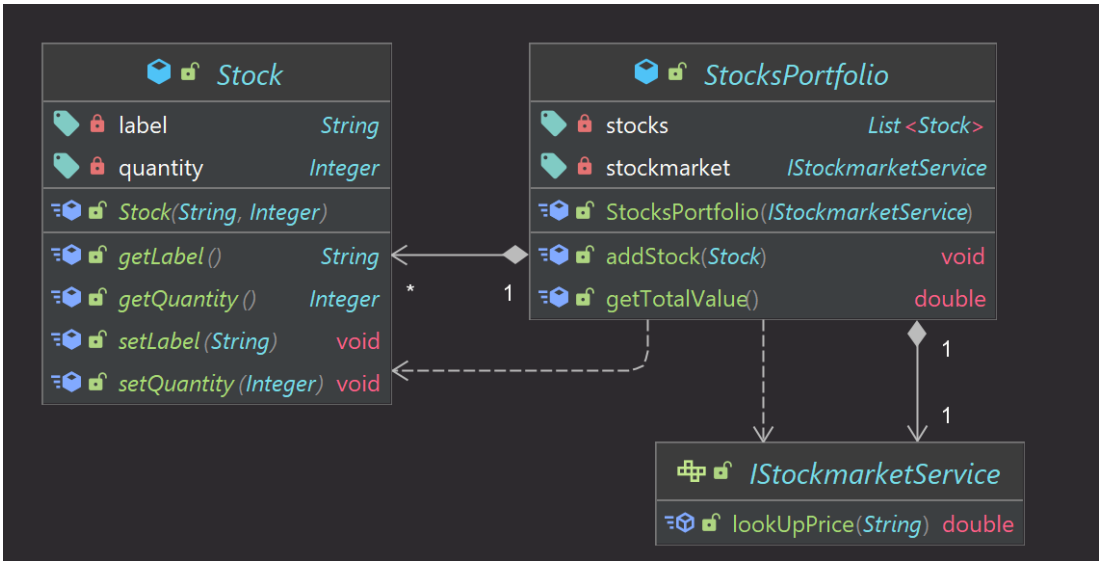


Figure 1: Classes for the StocksPortfolio use case.

a) Create the classes. You may write the implementation of the services before or after the tests.

b) Create the test for the getTotalValue(). As a guideline, you may adopt this outline:

```
1. Prepare a mock to substitute the remote service (@Mock annotation)
2. Create an instance of the subject under test (SuT) and use the mock to set
   the (remote) service instance.
3. Load the mock with the proper expectations (when...thenReturn)
4. Execute the test (use the service in the SuT)
5. Verify the result (assert) and the use of the mock (verify)
```

Notes:

— Consider use these Maven dependencies for your POM (JUnit5, Mockito).

— Mind the JUnit version. For JUnit 5, you should use the @ExtendWith annotation to integrate the Mockito framework.

```
@ExtendWith(MockitoExtension.class)
class StocksPortfolioTest { … }
```

— See an example of the main syntax and operations.

1b/ Instead of the JUnit core asserts, you may use the Hamcrest library to create more human-readable assertions. Replace the "Assert" statements in the previous example, to use Hamcrest constructs. See example.

2.2 Geocoding

Consider an application that needs to perform reverse geocoding to find a zip code for a given set of GPS coordinates. This service can be assisted by public APIs (e.g.: using the MapQuest API).
Let as create a simple application to perform (reverse) geocoding and set a few tests.

a) Create the objects represented in Figure 1. At this point, **do not implement TqsBasicHttpClient**; in fact, you should provide a substitute for it.

b) Consider that we want to test the behavior of AddressResolver#findAddressForLocation, which invokes a remote geocoding service, available in a REST interface, passing the site coordinates.

Which is the SuT (subject under test)? Which is the service to *mock*?

c) To create a test for findAddressForLocation, you will need to mimic **the exact (JSON) response of the geocoding service for a request**. Study/try the MapQuest API. See sample of response.

d) Implement a test for AddressResolver#findAddressForLocation (using mocks where required).

e) Besides the "success" case, consider also testing for alternatives (e.g.: bad coordinates;…). You may need to change the implementation.

This getting started project [gs-mockForHttpClient] can be used in your implementation.

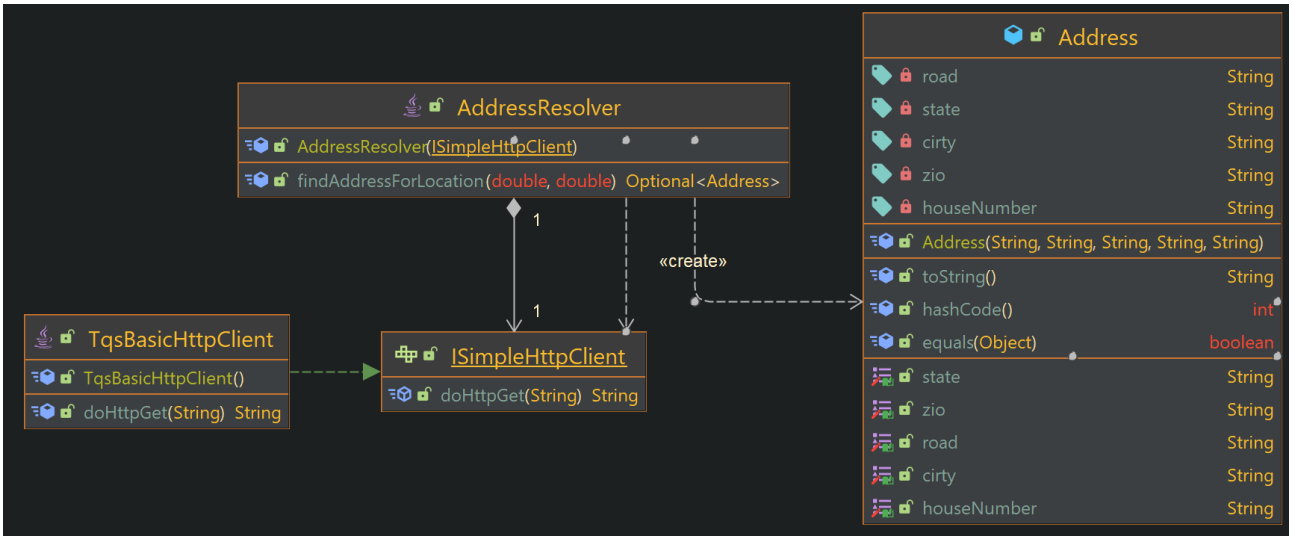


Figure 2: Classes for the geocoding use case.

2.3 Integration tests (with maven failsafe plugin)

Consider you are implementing an **integration test** for the previous example, and, in this case, you would use the real implementation of the module, not the mocks, in the test.
(This section can be included with the previous, continuing the same project.)

Create a test class (AddressResolver**IT**), in a separate test package, and be sure its name ends with "**IT**". Copy the tests from the previous exercise into this new test class but remove any support for mocking (no Mockito imports in this test).
Correct/complete the test implementation so it uses the real HttpClient implementation.
Run your test (and confirm that the remote API is invoked in the test execution).

Be sure the "failsafe" maven plugin is configured.
You should **get different results** with the following cases (try with and without internet connection):

```
$ mvn test
and
$ mvn install failsafe:integration-test
```

(Note the number of tests and the time required to run the tests…).

Lab 3 Multi-layer application testing (with Spring Boot)

Context and key points

Prepare
This lab is based on Spring Boot. Most of students already used the Spring Boot framework (in IES course). If you are new to Spring Boot, then you need to develop a basic understanding or collaborate with a colleague. Learning resources are available at the Spring site.

Key Points

- @SpringBootTest annotation loads whole application context, but it is better (faster) to limit application contexts only to a set of Spring components that participate in test scenario.
- @DataJpaTest only loads @Repository spring components, and will greatly improve performance by not loading @Service, @Controller, etc.
- Use @WebMvcTest to test Rest APIs exposed through Controllers. Beans used by controller need to be mocked.
- Isolate the functionality to be tested by limiting the context of loaded frameworks/components. For some use cases, you can even test with just standard unit testing.

Explore

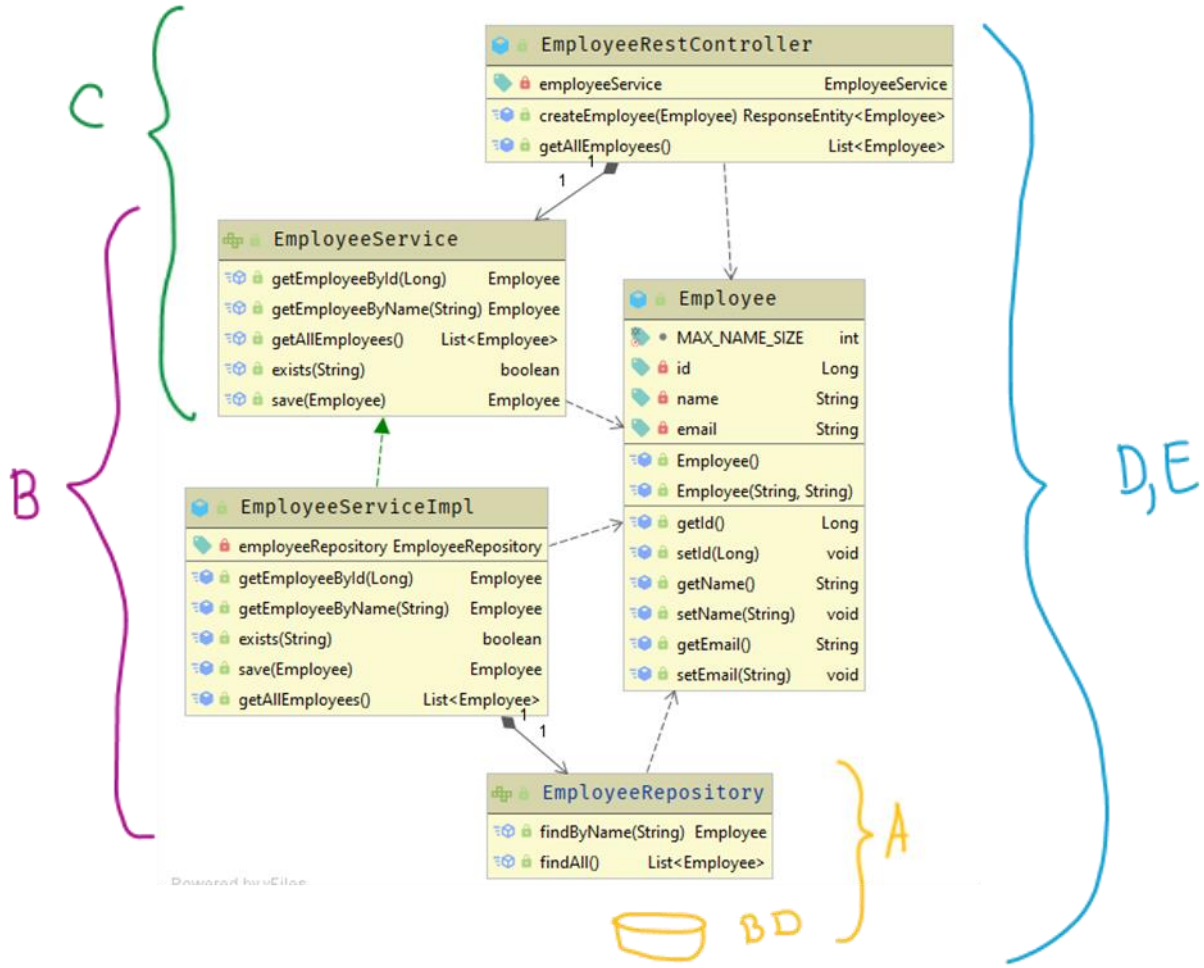
- AssertJ library to create expressive assertions in tests: https://assertj.github.io/doc/
- Talk on Spring Boot tests (by Pivotal): https://www.youtube.com/watch?v=Wpz6b8ZEgcU

3.1 Employee manager example

Study the example concerning a simplified Employee management application (project: gs-employee-manager).
This application follows the Spring Boot style to structure the solution:

- Employee: entity (@Entity) representing a domain concept.
- EmployeeRepository: the interface (@Repository) defining the data access methods on the target entity, based on the framework JpaRepository. “Standard” requests can be inferred and automatically supported by the framework; custom queries should be declared, if needed.
- EmployeeService and EmployeeServiceImpl: define the interface and its implementation (@Service) of a service related to the “business logic” of the application. Elaborated decisions/algorithms, for example, would be implemented in this component.
- EmployeeRestController: the component that implements the REST-endpoint/boundary (@RestController): handles the HTTP requests and delegates to the EmployeeService.

The project already contains a set of tests.



Look up and study the following test scenarios:

Purpose/scope	Strategy	Notes
A/ Verify the data access services provided by the repository component. [*EmployeeRepositoryTest*]	Slice the test context to limit to the data instrumentation (@DataJpaTest) Inject a TestEntityManager to access the database; use this object to write to the database directly (no caches involved).	@DataJpaTest includes the @AutoConfigureTestDatabase. If a dependency to an embedded database is available, an in-memory database is set up. Be sure to include H2 in the POM.
B/ Verify the business logic associated with the services implementation. [*EmployeeService_UnitTest*]	Often can be achieved with unit tests, given one mocks the repository. Rely on Mockito to control the test and to set expectations and verifications.	Relying only in JUnit + Mockito makes the test a unit test, much faster that using a full SpringBootTest. No database involved.
C/ Verify the boundary components (controllers); just the controller behavior. [*EmployeeController_ WithMockServiceTest*]	Run the tests in a simplified and light environment, simulating the behavior of an application server, by using @WebMvcTest mode. Get a reference to the server context with @MockMvc. To make the test more localized to the controller, you may mock the dependencies on the service (@MockBean); the repository component will not be involved.	MockMvc provides an entry point to server-side testing. Despite the name, is not related to Mockito. MockMvc provides an expressive API, in which methods chaining is expected. In principle, no database is involved.

D/ Verify the boundary components (controllers). Load the full Spring Boot application. No API client involved. [*EmployeeRestControllerIT*]	Start the full web context (@SpringBootTest, with Web Environment enabled). The API is deployed into the normal SpringBoot context. Use the entry point for server-side Spring MVC test support (MockMvc).	This would be a typical integration test in which several components will participate (the REST endpoint, the service implementation, the repository, and the database).
E/ Verify the boundary components (controllers). Load the full application. Test the REST API with explicit HTTP client. [*EmployeeRestControllerTemplateIT*]	Start the full web context (@SpringBootTest, with Web Environment enabled). The API is deployed into the normal SpringBoot context. Use a REST client to create realistic requests (TestRestTemplate)	Similar to the previous case, but instead of assessing a convenient servlet entry point for tests, uses an API client (so request and response un/marshaling will be involved).

Note 1: both D/ and E/ load the full Spring Boot Application (auto scan, etc…). The main difference is that in D/ one accesses the server context through a special testing servlet (MockMvc object), while in E/ the requester is a REST client (TestRestTemplate).

Note 2: you may run individual tests using maven command line options. E.g.:

```
$ mvn test -Dtest=EmployeeService*
```

Review questions: [answer in a **readme.md** file, in /lab3_1 folder]

a) Identify a couple of examples that use AssertJ expressive methods chaining.

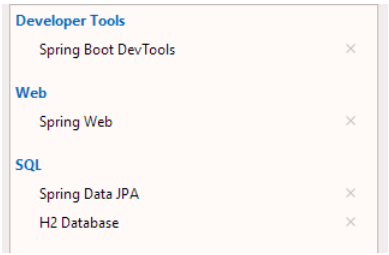
b) Identify an example in which you mock the behavior of the repository (and avoid involving a database).

c) What is the difference between standard @Mock and @MockBean?

d) What is the role of the file "application-integrationtest.properties"? In which conditions will it be used?

e) the sample project demonstrates three test strategies to assess an API (C, D and E) developed with SpringBoot. Which are the main/key differences?

3.2 Cars service



Consider the case in which you will develop an API for a car information system (as a Spring Boot application).
Consider using the Spring Boot Initializr to create the new project (either online or may be integrated in your IDE);
Add the dependencies (*starters*) for: Developer Tools, Spring Web, Spring Data JPA and H2 Database.

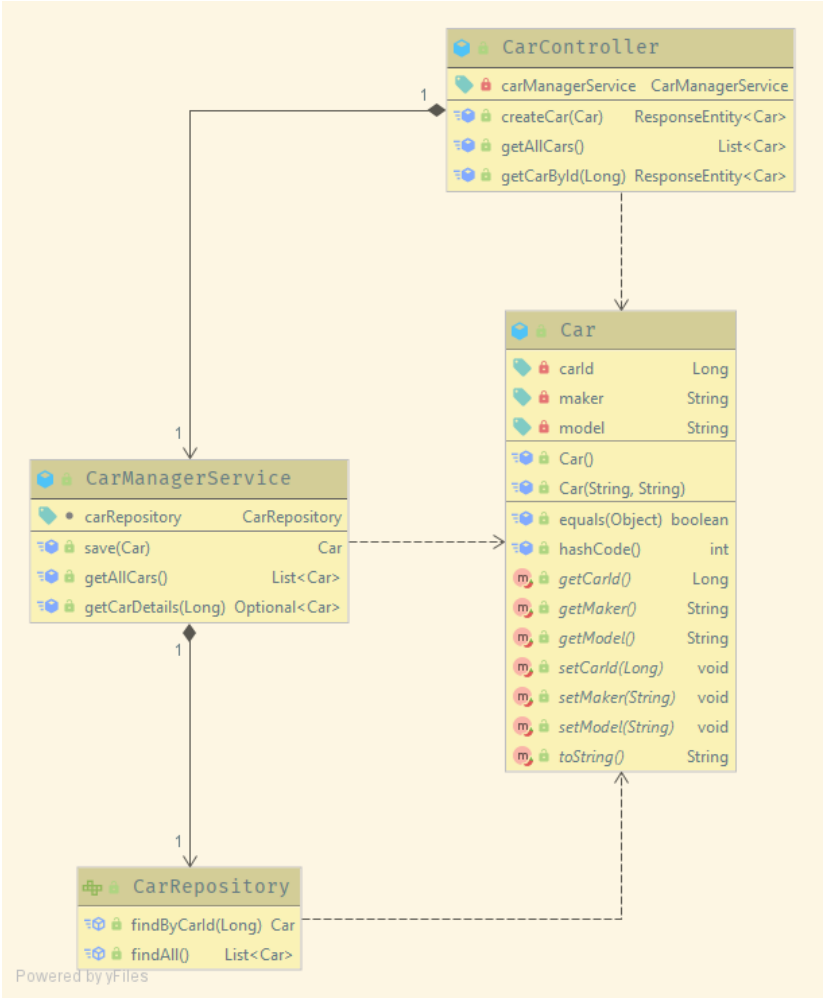
Use the structure modeled in the class diagram as a (minimal) reference.
In this exercise, **try to force a TDD approach**: write the test first; make sure the project can compile without errors; **defer the actual implementation of production code as much as possible**.
This approach will be encouraged if we try to write the tests in a top-down approach: **start from the controller, then the service, then the repository**.

a) Create a test to verify the Car [Rest]Controller (and mock the CarService bean), as "resource efficient" as possible. Run the test.

b) Create a test to verify the CarService (and mock the CarRepository). This can be a standard unit test with mocks.

c) Create a test to verify the CarRepository persistence. Be sure to include an in-memory database dependency in the POM (e.g.: H2).

d) Having all the previous tests passing, implement an integration test to verify the API. Suggestion: use the approach "**E/**" discussed in the previous project (Employees).



Note:
Although the Service in this example is quite trivial (just delegates to repository), in a larger application, we would expect the "services" to implement more complex, interesting, and mission-critical business logic, e.g.: "find a car that provides a *suitable replacement* for some given car [e.g.: as a courtesy car]".

3.3 Integration test

[Continue in the same project of the previous exercise.]
Adapt the integration test to use a real database. E.g.:

- Run a mysql instance and be sure you can connect (for example, using a Docker container)
- Change the POM to include a dependency to mysql [optionally remove H2].
- Add the connection properties file in the resources of the "test" part of the project (see the application-integrationtest.properties in the sample project)
- Use the @TestPropertySource and deactivate the @AutoConfigureTestDatabase.